From Grammars to Parameters: Automatic Iterated Greedy Design for the Permutation Flow-shop Problem with Weighted Tardiness

Franco Mascia, Manuel López-Ibáñez, Jérémie Dubois-Lacoste, and Thomas Stützle

IRIDIA, CoDE, Université Libre de Bruxelles, Brussels, Belgium
{fmascia,manuel.lopez-ibanez,jeremie.dubois-lacoste,stuetzle}@ulb.ac.be

Abstract. Recent advances in automatic algorithm configuration have made it possible to configure very flexible algorithmic frameworks in order to fine-tune them for particular problems. This is often done by the use of automatic methods to set the values of algorithm parameters. A rather different approach uses grammatical evolution, where the possible algorithms are implicitly defined by a context-free grammar. Possible algorithms may then be instantiated by repeated applications of the rules in the grammar. Through grammatical evolution, such an approach has shown to be able to generate heuristic algorithms. In this paper we show that the process of instantiating such a grammar can be described in terms of parameters. The number of parameters increases with the maximum number of applications of the grammar rules. Therefore, this approach is only practical if the number of rules and depth of the derivation tree are bounded and relatively small. This is often the case in the heuristic-generating grammars proposed in the literature, and, in such cases, we show that the parametric representation may lead to superior performance with respect to the representation used in grammatical evolution. In particular, we first propose a grammar that generates iterated greedy (IG) algorithms for the permutation flow-shop problem with weighted tardiness minimization. Next, we show how this grammar can be represented in terms of parameters. Finally, we compare the quality of the IG algorithms generated by an automatic configuration tool using the parametric representation versus using the codon-based representation of grammatical evolution. In our scenario, the parametric approach leads to significantly better IG algorithms.

Keywords: automatic algorithm configuration, grammatical evolution, iterated greedy, permutation flow-shop problem

1 Introduction

Designing an effective stochastic local search (SLS) algorithm for a hard optimisation problem is a time-consuming, creative process that relies on the experience and intuition of the algorithm designer. Recent advances in automatic algorithm

configuration have shown that this process can be partially automated, thus reducing human effort. This allows algorithm designers to explore a larger number of algorithm designs than it was previously feasible and, by relying less on human intuition, to explore many design choices that would have never been implemented and tested because they were regarded as supposedly poor design alternatives.

Nowadays, methods for automatic algorithm configuration are able to handle large parameter spaces composed of both categorical and numerical parameters with complex interactions. This capability has enabled researchers to automatically configure flexible frameworks of state-of-the-art SAT solvers [7], and to automatically design multi-objective algorithms [9]. The development of these flexible frameworks follows a *top-down* approach, in which a framework for generating SLS algorithms is built starting from algorithmic components already known to perform well for the problem at hand.

Instead, we consider here a *bottom-up* approach, where an algorithm is assembled from simple components without a priori fixing how they could be combined. There are two recent works that follow such a bottom-up approach. Vázquez-Rodríguez and Ochoa [13] automatically generate by using genetic programming an initial order for the NEH algorithm, a well-known constructive heuristic for the PFSP. More recently, Burke et al. [2] automatically generate iterated greedy (IG) algorithms for the one-dimensional bin packing problem. Both works instantiate algorithms bottom-up from a context-free grammar.

In this paper, we propose to use a parametric representation, using categorical, numerical and conditional parameters, to instantiate algorithms from a grammar. In particular, we show how grammars can be represented in terms of a parametric space, using categorical, numerical and conditional parameters. Such parametric representation exploits the abilities of automatic configuration tools, which are mentioned above. Moreover, the proposed parametric representation avoids known disadvantages of GE, such as low fine-tuning behaviour due to the low locality of the operators used by GE [10]. We apply our proposed approach to the automatic generation of IG algorithms for the permutation flow-shop problem (PFSP). The PFSP models many variants of a common kind of production environment in industries. Because of its relevance in practice, the PFSP has attracted a large amount of research since its basic version was formally described decades ago [6]. Moreover, with the exception of few special cases, most variants of the PFSP are $\mathcal{NP}$-hard [5], and, hence, tackling real-world instances often requires the use of heuristic algorithms. For these reasons, the PFSP is an important benchmark problem for the design and comparison of heuristics. When tackling new PFSP variants, the automatic generation of heuristics can save a significant effort.

Finally, we also compare our proposed parametric representation with the codon-based representation used in GE. Our experiments show that, for the particular grammar considered in this paper, the parametric representation produces better heuristics than the GE representation.

This paper is structured as follows. In Section 2 we introduce the PFSP problem we tackle. Section 3 presents the methodology we use and describes the mapping from a grammar to a parametric representation. Next, we present our experimental results in Section 4 and we conclude in Section 5.

2 Permutation Flowshop Scheduling

The flowshop scheduling problem (FSP) is one of the most widely studied scheduling problems, as it models a very common kind of production environment in industries. The goal in the FSP is to find a schedule to process a set of n jobs $(J_1, \ldots, J_n)$ on m machines $(M_1, \ldots, M_m)$. The specificity of flowshop environments is that all jobs must be processed on the machines in the same order, i.e., all jobs have to be processed on machine M_1, then machine M_2, and so on until machine M_m. A common restriction in the FSP is to forbid job passing between machines, i.e., to restrict to solutions that are permutations of jobs. The resulting problem is called permutation flowshop scheduling problem (PFSP). In the PFSP, all processing times p_{ij} for a job J_i on a machine M_j are fixed, known in advance, and non-negative. In what follows, C_{ij} denotes the completion time of a job J_i on machine M_j and C_i denotes the completion time of a job J_i on the last machine, M_m.

In many practical situations, for instance when products are due to arrive at customers at a specific time, jobs have an associated *due date*, denoted here by d_i for a job J_i. Moreover, some jobs may be more important than others, which can be expressed by a weight associated to them representing their priority. Thus, the so-called *tardiness* of a job J_i is defined as $T_i = \max\{C_i - d_i, 0\}$ and the *total weighted tardiness* is given by $\sum_{i=1}^{n} w_i \cdot T_i$, where w_i is the priority assigned to job J_i.

We consider the problem of minimizing the total weighted tardiness (WT). This problem, which we call *PFSP-WT*, is $\mathcal{NP}$-hard in the strong sense even for a single machine [3]. Let π_i denote the job in the i-th position of a permutation π. Formally, the *PFSP-WT* consists of finding a given job permutation π as to

$$
\begin{aligned}
\text{minimize} \quad & F(\pi) = \sum_{i=1}^{n} w_i \cdot T_i \\
\text{subject to} \quad & C_{\pi_0 j} = 0 \quad j \in \{1, \ldots, m\}, \\
& C_{\pi_i 0} = 0 \quad i \in \{1, \ldots, n\}, \\
& C_{\pi_i j} = \max\{C_{\pi_{i-1} j}, C_{\pi_i j-1}\} + p_{ij}, \\
& \quad i \in \{1, \ldots, n\} \quad j \in \{1, \ldots, m\}. \\
& T_i = \max\{C_i - d_i, 0\} \quad i \in \{1, \ldots, n\}.
\end{aligned}
\tag{1}
$$

We tackle the *PFSP-WT* by means of iterated greedy (IG), which has been shown to perform well in several PFSP variants [11]. Our goal is to *automatically* generate IG algorithms in a bottom-up manner from a grammar description. The next section explains our approach in more detail.

3 Methods

The methodology used in this work is the following. Given a problem to be tackled, we first define a set of algorithmic components for the problem at hand, avoiding as much as possible assumptions on which component will contribute the most to the effectiveness of the algorithm or which is the best way of combining the components in the final algorithm. Once the building blocks are defined, we use tools for automatic algorithm configuration to explore the large design space of all possible combinations and select the best algorithm for the problem at hand.

The size and complexity of the building blocks is set at a level that is below a full-fledged heuristic, but still allows us to easily combine them in a modular way to generate a very large number of different algorithms. This is in contrast with the more standard way of designing SLS algorithms for a given problem, in which the algorithm designer defines the full-fledged heuristics and leaves out some parameters to tune specific choices within the already defined structure.

In this paper, the building blocks and the way in which they can be combined will be described by means of context-free grammars. Grammars comprise a set of rules that describe how to construct sentences in a language given a set of symbols. The grammar discussed in this paper generates algorithm descriptions in pseudo-code, the actual grammar used in the experiments is equivalent to the one presented in this paper but generates directly C++ code.

3.1 The Grammar for PFSP

The PFSP has been the object of many studies over the past decades, and it is still attracting a significant amount of research nowadays. Recent studies [11, 4] have shown that many high-performing algorithms to tackle the PFSP (whatever is the objective to optimize) are based on the *iterated greedy* (IG) principle. IG consists of the iterative partial "destruction" of the current solution, and its "reconstruction" into a full solution afterwards. The term "greedy" comes from the fact that the reconstruction of the solution is often done using a greedy heuristic. In the case of the PFSP, the destruction phase removes a number of jobs from the schedule. The reconstruction phase inserts these jobs back in the solution, to obtain again a complete solution.

In this work, we define a grammar for generating IGs for the PFSP in which we allow several underlying heuristic choices. State-of-the-art IG algorithms for the PFSP from the literature always apply a local search step to each full solution after the reconstruction phase. However, such local search step would hide performance differences when using different choices for the other components of IG. Our goal is not to generate a state-of-the-art IG for the PFSP, but rather to study different methods for the automatic generation of algorithms, and, thus, in this paper we do not apply any local search step.

Fig. 1 shows the grammar for generating the main step of the IG algorithm in Backus–Naur Form (BNF). In BNF, production rules are in the form `<non-terminal> ::= expression`. Each rule describes how the non-terminal

```
<program> ::= procedure ig_step()
                  <select_jobs>
                  <select_more>
                  remove_selected()
                  sort_removed_jobs(<order_criteria> <tie_breaking>)
                  insert_jobs(insert_criteria)

<select_more> ::= <select_jobs> <select_more>
                | ""

<select_jobs> ::= select_jobs(<heuristic>, <num>, <low_range>, <high_range>)

  <heuristic> ::= priority | position | sumProcessingTimes | dueDate
                | tardiness | waitingTime | idleTime

        <num> ::= [0..100]

  <low_range> ::= [0..99]

 <high_range> ::= [0..100]

<tie_breaking> ::= , <order_criteria> <tie_breaking>
                | ""

<order_criteria> ::= order(<comparator>, <heuristic>)

 <comparator> ::= "<" | ">"

<insert_criteria> ::= weightedTardiness
                    | weightedTardiness, sumCompletionTimes
                    | weightedTardiness, sumCompletionTimes, weightedEarlyness
                    | weightedTardiness, weightedEarlyness
                    | weightedTardiness, weightedEarlyness, sumCompletionTimes
```

Fig. 1. Grammar that describes the rules for generating IG algorithms for the PFSP.

symbol on the left-hand side can be replaced by the expression on the right-hand side. Expressions are strings of terminal and/or non-terminal symbols. If there are alternative strings of symbols for the replacement of the non terminal on the left-hand side, the alternative strings are separated with the symbol "`|`".

In Fig. 1, the non-terminal symbol `<program>` defines the main step of the algorithm. First one or more jobs are marked for removal from the current solution, then the selected jobs are removed and sorted, and finally the solution is reconstructed inserting the jobs back in the current solution.

Implementing an IG algorithm for the PFSP requires to make some design choices. In particular, (i) which jobs and how many are selected for removal, (ii) in which order the jobs are reinserted; and (iii) which criteria should be optimized when deciding the insertion point. All the possibilities that we consider in this paper are described by the grammar in Fig. 1. Next, we explain these components in detail.

Heuristics for the selection of jobs. The selection of jobs for removal (rule `<select_jobs>`) consists in the application of one or more selection rules. In particular this is done with the function `select_jobs(<heuristic>, <num>, <low_range>, <high_range>)` that selects `<num>` jobs from the current solution

according to the rule specified in `<heuristic>`. Each rule computes a numerical value for each job J_i, which may be one of the following properties:

- `Priority`: the weight w_i that defines its priority;
- `DueDate`: its due date d_i;
- `SumProcessingTimes`: the sum of its processing times, $\sum_{j=1}^{m} p_{ij}$;
- `Position`: its position in the current solution;
- `Tardiness`: its tardiness in the current solution;
- `WaitingTime`: its waiting time between machines computed as $\sum_{j=2}^{m} C_{\pi_i j} - C_{\pi_i j-1} - p_{\pi_i j}$;
- `IdleTime`: the time during which machines are idle because the job is still being processed on a previous machine, that is, $\sum_{j=1}^{m} C_{\pi_i j} - C_{\pi_{i-1} j} - p_{\pi_i j}$, for $i \neq \pi_1$.

After the heuristic values are computed, they are normalized in the following way: the minimum for each heuristic value among all jobs is normalized to 0, the maximum one to 100, and values in-between are normalized linearly to the range $[0, 100]$. Only jobs whose normalized heuristic value is between a certain range $[low, high]$ are considered for selection. The range is computed from the values given by the grammar as $high =$ `<high_range>` and $low =$ `<low_range>` $\cdot\, high/100$. Finally, from the jobs considered for selection, at most `<num>` percent (computed as `<num>` $\cdot\, n/100$) of the jobs are actually selected, where n is the total number of jobs. An example of selection rule would be `select_jobs(DueDate,20,10,50)`, which means that, from those jobs that have a normalized due date in the range $[10, 50]$, at most $0.2 \cdot n$ jobs are selected.

Rules for ordering the jobs. The function that sorts jobs for re-insertion (`sort_removed_jobs`) is composed by one or more order criteria (`<order_criteria>`), where each additional order criterion is used for breaking ties. Each order criterion sorts the removed jobs by a particular heuristic value, in either ascending or descending order, according to `<comparator>`. The result is a permutation of the removed jobs according to the order criteria.

Rules for inserting the jobs. In this paper we consider the minimization of the weighted tardiness of the solution, thus, it is natural to optimize primarily this objective when choosing the position for re-inserting each job. However, it often happens that the weighted tardiness is the same for any insertion position of a job (in particular, when the solution is partial: all jobs can easily respect due dates and therefore the weighted tardiness is 0).

Thus, we consider the possibility of breaking ties according to additional criteria, namely, the minimization of the *sum of completion times* and the maximization of the *weighted earliness*, computed as $\sum_{i=1}^{n} w_i \cdot (d_i - C_i)$. Both are correlated with the minimization of the weighted tardiness and allow us to differentiate between partial schedules with zero weighted tardiness because none of the jobs is tardy. In total, we consider five alternatives for the insertion criteria (`<insert_criteria>`), corresponding to breaking ties with any combination of either, none or both sum of completion times and weighted earliness.

Table 1. Parametric representation of the grammar in Fig. 1

Parameter	Domain	Condition
select_jobs$_1$	{Priority, Position, SumProcessingTimes, DueDate, Tardiness, WaitingTime, IdleTime}	
num$_1$	[0,100]	
low_range$_1$	[0,99]	
high_range$_1$	[0,100]	
select_jobs$_2$	{Priority, Position, SumProcessingTimes, DueDate, Tardiness, WaitingTime, IdleTime, ""}	
num$_2$	[0,100]	if select_jobs$_2 \neq$ ""
low_range$_2$	[0,99]	if select_jobs$_2 \neq$ ""
high_range$_2$	[0,100]	if select_jobs$_2 \neq$ ""
...	...	
select_jobs$_i$	{Priority, Position, SumProcessingTimes, DueDate, Tardiness, WaitingTime, IdleTime, ""}	if select_jobs$_{i-1} \neq$ ""
num$_i$	[0,100]	if select_jobs$_i \neq$ ""
low_range$_i$	[0,99]	if select_jobs$_i \neq$ ""
high_range$_i$	[0,100]	if select_jobs$_i \neq$ ""
order_criteria$_1$	{Priority, Position, SumProcessingTimes, DueDate, Tardiness, WaitingTime, IdleTime}	
comparator$_1$	{"<", ">"}	
order_criteria$_2$	{Priority, Position, SumProcessingTimes, DueDate, Tardiness, WaitingTime, IdleTime, ""}	
comparator$_2$	{"<", ">"}	if order_criteria$_2 \neq$ ""
...	...	
order_criteria$_j$	{Priority, Position, SumProcessingTimes, DueDate, Tardiness, WaitingTime, IdleTime, ""}	if order_criteria$_{j-1} \neq$ ""
comparator$_j$	{"<", ">"}	if order_criteria$_j \neq$ ""
insert_criteria	{"WaitingTime", "WaitingTime, SumCompletionTimes", "WaitingTime, SumCompletionTimes, WeightedEarlyness", "WeightedEarlyness, WeightedEarlyness", "WaitingTime, WeightedEarlyness, SumCompletionTimes"}	

3.2 From Grammars to Parameters

To tune the algorithms with a tool for automatic algorithm configuration, we need to define the process of instantiating a grammar as a choice between alternative parameter settings. Table 1 is a possible parametric representation of the grammar given in Fig. 1. We now explain in detail how the parametric representation was obtained.

First, rules that do not contain alternatives do not require a parameter. Second, numeric terminals, such as `<num>`, `<low_range>` and `<high_range>` in Fig. 1 can be naturally represented by numerical parameters with a defined range. Third, rules with alternative choices are represented as categorical parameters. This is especially natural in the case of rules that consist only of alternative terminals, such as `<insert_criteria>`.

The only difficulty appears if the same rule can be applied more than once, for example, rules `<select_jobs>` and `<tie_breaking>`. In such a case, each application requires its own parameter. Some of these rules might be applied an infinite number of times, and, thus, they might seem to require an infinite number of parameters. However, when generating algorithms from grammars, such rules are never applied more than a small number of times. We use this

consideration and explicitly limit the number of parameters that describe such rules; thus, in this way we also limit the length of the generated algorithm.

Converting rules that can be derived an unbounded number of times is the non trivial case, and we will explain it here with an example. Assume we want to map the following rule to a set of categorical parameters:

```
<select_jobs> ::= <a_job> | <a_job> <select_jobs>
      <a_job> ::= criterion1 | criterion2
```

What is expressed by the rule is that a valid program contains a list of at least one criterion. Suppose we want to limit the number of rule-applications to five, then the rule could be converted into five categorical parameters with possible values `criterion1` or `criterion2`. This mapping leads to exactly five criteria. To have *at most* five, the parameters should consider also the empty string among the possible values. The corresponding grammar would be the following:

```
<select_jobs> ::= <a_job> <a_job> <a_job> <a_job> <a_job>
      <a_job> ::= criterion1 | criterion2 | ""
```

In order to have *at least* one job, the first parameter should not have the empty string among the possible values. This would more directly map to the following equivalent grammar:

```
<select_jobs>  ::= <a_job> <further_jobs>
<further_jobs> ::= "" | <a_job> <further_jobs>
      <a_job>  ::= criterion1 | criterion2
```

Table 1 shows the mapping of Fig. 1 to parameters. Both rules `<select_jobs>` and `<order_criteria>` can be applied up to i and j times respectively. Moreover, each parameter used in those rules has to be duplicated for each possible application of the rules.

3.3 From Grammars to Sequences of Integers

How to search for the best algorithm in the design space defined by the grammar and how to represent the sequence of derivation rules that represent an algorithm is the goal of different methods in grammar based genetic programming (GBGP) [10]. Among the GBGP techniques proposed in the literature, we consider recent works in grammatical evolution (GE) [2].

In GE, the instantiation of a grammar is done by starting with the `<program>` non-terminal symbol, and successively applying the derivation rules in the grammar, until there are no non-terminal symbols left. Every time that a non-terminal symbol can be replaced following more than one production rule, a choice has to be made. The sequence of specific choices made during the derivation, which leads to a specific program, is encoded in a sequence of integers.

This linearisation of the derivation tree, leads to a high decoupling between the sequence of integers and the programs being generated. For example, when

a derivation is complete and there are still numbers left in the sequence, these numbers are discarded. Conversely, if the derivation is not complete and there are no numbers left in the sequence, the sequence is read again from the beginning. This operation is called wrapping and is repeated for a limited number of times. If after a given number of wrappings the derivation is not complete, the sequence of strings is considered to lead to an invalid program. Moreover, since the integers are usually in a range which is bigger than the possible choices for the derivation of a non terminal, a modulo operation is applied at each choice.

In GE, the sequences of integers are used as chromosomes in a genetic algorithm that is used to derive the best algorithm for a given problem. The high decoupling between the programs and their representation, has it drawbacks when used within a genetic algorithm. The decoupling translates to non locality in the mutation and crossover operators [10]. Wrapping operations are clearly responsible of this decoupling, but even without wrapping, the way in which an algorithm is derived from a grammar and the sequence of integer values leads to non locality in the operation. In fact, since the integer values are used to transform the left-most non terminal symbol, a choice in one of the early transformations can impact on the structure of the program being generated and on the meaning of all subsequent integers in the sequence. Therefore since a mutation on one integer in the sequence (a codon) impacts on the meaning of all the following codons, one-bit mutations in different positions of the individual genotype have impacts of different magnitude on the phenotype of the individual. For the same reason the offspring of two highly fit parents is not necessarily composed of highly fit individuals. On the contrary a one-point cross-over of the best individuals in the population could lead to individuals whose genotype can not be translated to any algorithm, because of the upper-bound on the wrapping operations. But, regardless of the specific issues when used in a genetic algorithm, we are interested to see if this representation presents similar drawbacks also when used with a tool for automatic algorithm configuration. In fact, this linearisation of the grammar, can easily be used within a tool for algorithmic configuration by mapping all codons to categorical parameters. The choice here between integer and categorical parameters is due to the high non linear response between the values of the codons and the algorithm they are decoded into.

Both the parameters and the sequence of codons limit the length of the algorithms that can be generated. In fact, a grammar can represent an arbitrarily long algorithm, but in practice the length is limited by the number of parameters in one case, and in the other case by the number of possible wrapping operations.

4 Experimental Results

4.1 Experiments

The automatic configuration procedure used in this work is irace [8], a publicly available implementation of Iterated F-Race [1]. Iterated F-Race starts by sampling a number of parameter configurations uniformly at random. Then, at each

iteration, it selects a set of elite configurations using a racing procedure and the non-parametric Friedman test. This racing procedure runs the configurations iteratively on a sequence of (training) problem instances, and discards configurations as soon as there is enough statistical evidence that a configuration is worse than the others. After the race, the elite configurations are used to bias a local sampling model. The next iteration starts by sampling new configurations from this model, and racing is applied to these configurations together with the previous elite configurations. This procedure is repeated until a given budget of runs is exhausted. The fact that irace handles categorical, numerical and surrogate parameters with complex constraints makes it ideal to instantiate algorithms from grammars in the manner proposed in this paper.

Benchmark Sets. We generated two benchmark sets of PFSP instances: 100 instances of 50 jobs and 20 machines (`50x20`), and 100 other instances of 100 jobs and 20 machines (`100x20`). These two sizes are nowadays the most common ones in the literature to evaluate heuristic algorithms on various PFSP variants. The processing times of the jobs on each machine are drawn from a discrete uniform distribution $\mathcal{U}\{1,\ldots,99\}$ [12]. The weights of the jobs are generated at random from $\mathcal{U}\{1,\ldots,10\}$, and each due date d_i is generated in a range proportional to the sum of processing times of the job J_i as: $d_i = \left\lfloor r \cdot \sum_{j=1}^{m} p_{ij} \right\rfloor$, where r is a random number sampled from the continuous uniform distribution $\mathcal{U}(1,4)$.

Experimental Setup. We compare the quality of the heuristics generated by irace when using either the grammar representation used by GE (`irace-ge`) or the parametric representation given in Table 1 (`irace-param`). In `irace-ge` an algorithm is derived from the grammar by means of 30 codons, which are mapped to 30 integer parameters that can assume values in the range $[0, 100]$. For the parametric representation given in Table 1, we need to specify the number of times the `select_jobs` and `order_criteria` rules are applied (i and j, respectively). Large values give more flexibility to the automatic configuration tool to find the best heuristics, however, they also enlarge the space of potential heuristics. We study three possibilities: `irace-param5`, which uses $i = 5$, $j = 3$; `irace-param3`, which uses $i = 3$, $j = 3$, and `irace-param1`, which uses $i = 1$, $j = 1$. The first variant is larger than what we expect to be necessary, and its purpose is to test if irace can find shorter heuristics than the maximum bounds. The purpose of the last variant is to verify that more than one application per rule is necessary to generate good results.

Each run of irace has a maximum budget of 2 500 runs of IG, and each run of IG is stopped after $0.001 \cdot n \cdot m$ seconds.

Using the same computational budget, we also consider two additional methods that generate heuristics randomly, to use as a baseline comparison. These methods generate 250 IG heuristics randomly, run them on 10 randomly selected training instances and select the heuristic that obtains the lowest mean value. Method `rand-ge` uses the grammar representation, while method `rand-param` uses the parametric representation.

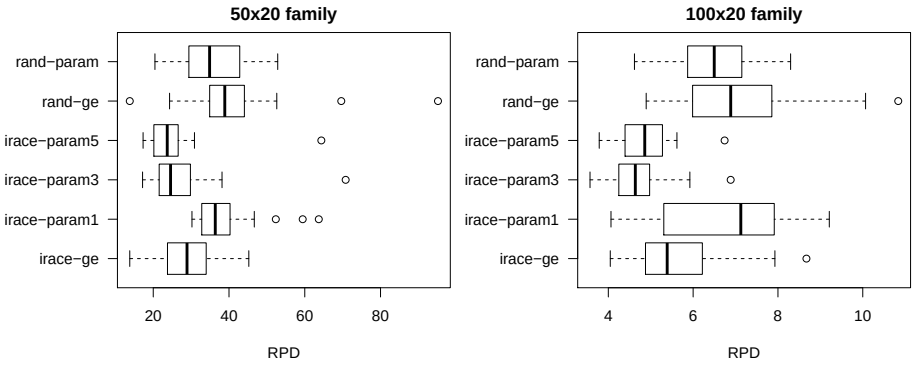

Fig. 2. Mean relative percentage deviation (RPD) obtained by the heuristics generated by each tuning method. Results are given separately for the heuristics trained and tested on `50x20` instances and on `100x20` instances.

Each method (`irace-ge`, `irace-param5`, `irace-param3`, `irace-param1`, `rand-ge`, `rand-param`) is repeated 30 times with different random seeds for each benchmark set, that is, in total, each method generates 60 IG heuristics. The training set used by all methods are the first 90 instances of each size. A grammar equivalent to the one in Fig. 1 is used to generate directly C++ code, which is in turn compiled with GCC 4.4.6 with optimization level `-O3`. Experiments were run on a single core of an AMD Opteron 6272 CPU (2.1 GHz, 16 MB L2/L3 cache size) running under Cluster Rocks Linux version 6/CentOS 6.3, 64bits.

4.2 Results

For assessing the quality of the generated heuristics, we run them on 10 test instances (that are distinct from the ones used for the training), repeating each run 10 times with different random seeds. Next, we compute the relative percentage deviation (RPD) from the best solution ever found by any run for each instance. The RPD is averaged over the 10 runs and over the 10 test instances.

Figure 2 compares the quality of the heuristics generated by the four methods described above on each test set for `50x20` and `100x20` benchmark sets. For each method, we plot the distribution of the mean RPD of each heuristic generated by it. In both benchmark sets, `irace-param5` and `irace-param3` obtain the best heuristics. The heuristics generated by `irace-param1` are typically worse than those generated by `irace-ge`.

Pairwise comparisons using the Wilcoxon signed rank test indicate that all pair-wise differences are statistically significant with the only exception being the pair `rand-param` and `irace-param1`. Moreover, we compare the different methods using the Friedman test in Table 2. Both `irace-param5` and `irace-param3` are ranked much better than `irace-ge`, thus confirming the superiority of the parameterised representation in our case studies.

Table 2. Comparison of the methods through the Friedman test on the two benchmark sets. $\Delta R_{\alpha=0.95}$ gives the minimum difference in the sum of ranks between two methods that is statistically significant. For both benchmark sets, `irace-param5` and `irace-param3` are clearly superior to the other methods.

Family	$\Delta R_{\alpha=0.95}$	Method (ΔR)
`50x20`	265.55	irace-param5 (0), irace-param3 (453), irace-ge (1574.5), rand-param (3706.5), irace-param1 (3976), rand-ge (4705)
`100x20`	262.85	irace-param3 (0), irace-param5 (474.5), irace-ge (2106), irace-param1 (4214.5), rand-param (4380), rand-ge (4917)

When analysing the heuristics generated by each method, we observe that both `irace-ge` and `rand-ge` generate on average around three `<select_jobs>` rules and no more than 2.23 `<order_criteria>` rules. On the other hand, `irace-param5` and `rand-param` generate on average more than 4.5 `<select_jobs>` rules and more than 2.5 `<order_criteria>` rules. This suggests to us that the GE-based representation has trouble generating programs with as many rules as the parametric representation. The results obtained by `irace-param1` and `irace-param3` suggest also that at least three rules are necessary to obtain good results. In terms of the particular heuristics generated, we observe that most heuristics contain a rule that selects jobs according to idle time. Perhaps more surprising is that the most common order criteria for sorting removed jobs is by position. On the other hand, there is no clear winner among the `insert_criteria` methods, which suggests that breaking ties in some particular order is not advantageous. A complete analysis of the heuristics is not the purpose of this paper, but our results indicate that the heuristics generated by irace are quite different from what a human expert would consider when designing a similar algorithm.

5 Conclusions

The main conclusion from our work is that existing automatic configuration tools may be used to generate algorithms from grammars.

Defining algorithmic components to be combined in an SLS algorithm presents several advantages over the design of a full fledged heuristic, where some design choices are left to be tuned automatically. Most importantly, less intuition, and therefore less bias of the designer goes in the definition of the separate blocks with respect to the classical top-down approach. But there are also more practical advantages of following a bottom-up strategy. In fact, every instantiation of the grammar is a minimal SLS algorithm designed and implemented to have a very specific behaviour. Less programming abstractions are needed, and a simpler code may be optimised more easily by the compilers. Even the parameters become constant values in the source code, and there is no need to pass them to various parts of the algorithm. On the contrary, when following a top-down approach, the designer tackles the hard engineering task of designing a full-fledged

framework where all possible combinations of design choices have to be defined beforehand. This leads to a reduced number of possible combinations with respect to a modular bottom-up approach, and also to the added complexity of intricate conditional expressions required to instantiate only the parts of the framework needed to express a specific algorithm.

We have shown that it is possible to represent the instantiation of the grammar by means of a parametric space. The number of parameters required is proportional to the number of times a production rule can be applied, and, hence, our approach is more appropriate for grammars where this number is bounded and not excessively large. It is an open research question for which kind of grammars the number of parameters required to represent applications of production rules becomes prohibitively expensive and other representations are more appropriate. Nonetheless, the grammar used in this work is similar in this respect to others that can be found in the literature, and, hence, we believe that grammars, where production rules are to be applied only a rather limited number of times are common in the development of heuristic algorithms.

From our experimental results, the heuristics generated by irace when using the parametric representation achieve better results than those generated when using the GE representation. This indicates that the parametric representation can help to avoid disadvantages of grammatical evolution such as a low fine-tuning behaviour due to a low locality of the used operators. Furthermore, our approach is not limited to irace and it can be applied using other automatic configuration tools, as long as they are able to handle categorical and conditional parameters.

In future work, we plan to compare our approach with a pure GE algorithm, which is the algorithm used in previous similar works. Moreover, our intention is to test the proposed method on different grammars and benchmark problems to investigate its benefits and limitations.

Acknowledgments. This work was supported by the META-X project, an *Action de Recherche Concertée* funded by the Scientific Research Directorate of the French Community of Belgium. Franco Mascia, Manuel López-Ibáñez and Thomas Stützle acknowledge support from the Belgian F.R.S.-FNRS, of which they are postdoctoral researchers and a research associate, respectively. Jérémie Dubois-Lacoste acknowledges support from the MIBISOC network, an Initial Training Network funded by the European Commission, grant PITN–GA–2009–238819. The authors also acknowledge support from the FRFC project "*Méthodes de recherche hybrids pour la résolution de problèmes complexes*". This research and its results have also received funding from the COMEX project within the Interuniversity Attraction Poles Programme of the Belgian Science Policy Office.

References

1. Balaprakash, P., Birattari, M., Stützle, T.: Improvement strategies for the F-race algorithm: Sampling design and iterative refinement. In: Bartz-Beielstein, T.,

Blesa, M.J., Blum, C., Naujoks, B., Roli, A., Rudolph, G., Sampels, M. (eds.) Hybrid Metaheuristics, Lecture Notes in Computer Science, vol. 4771, pp. 108–122. Springer, Heidelberg, Germany (2007)
2. Burke, E.K., Hyde, M.R., Kendall, G.: Grammatical evolution of local search heuristics. IEEE Transactions on Evolutionary Computation 16(7), 406–417 (2012)
3. Du, J., Leung, J.Y.T.: Minimizing total tardiness on one machine is NP-hard. Mathematics of Operations Research 15(3), 483–495 (1990)
4. Dubois-Lacoste, J., López-Ibáñez, M., Stützle, T.: A hybrid TP+PLS algorithm for bi-objective flow-shop scheduling problems. Computers & Operations Research 38(8), 1219–1236 (2011)
5. Garey, M.R., Johnson, D.S., Sethi, R.: The complexity of flowshop and jobshop scheduling. Mathematics of Operations Research 1, 117–129 (1976)
6. Johnson, D.S.: Optimal two- and three-stage production scheduling with setup times included. Naval Research Logistics Quarterly 1, 61–68 (1954)
7. KhudaBukhsh, A.R., Xu, L., Hoos, H.H., Leyton-Brown, K.: SATenstein: Automatically building local search SAT solvers from components. In: Boutilier, C. (ed.) Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI-09). pp. 517–524. AAAI Press/International Joint Conferences on Artificial Intelligence, Menlo Park, CA (2009)
8. López-Ibáñez, M., Dubois-Lacoste, J., Stützle, T., Birattari, M.: The irace package, iterated race for automatic algorithm configuration. Tech. Rep. TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium (2011)
9. López-Ibáñez, M., Stützle, T.: The automatic design of multi-objective ant colony optimization algorithms. IEEE Transactions on Evolutionary Computation 16(6), 861–875 (2012)
10. Mckay, R.I., Hoai, N.X., Whigham, P.A., Shan, Y., O'Neill, M.: Grammar-based genetic programming: a survey. Genetic Programming and Evolvable Machines 11(3-4), 365–396 (Sep 2010)
11. Ruiz, R., Stützle, T.: A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. European Journal of Operational Research 177(3), 2033–2049 (2007)
12. Taillard, É.D.: Benchmarks for basic scheduling problems. European Journal of Operational Research 64(2), 278–285 (1993)
13. Vázquez-Rodríguez, J.A., Ochoa, G.: On the automatic discovery of variants of the NEH procedure for flow shop scheduling using genetic programming. Journal of the Operational Research Society 62(2), 381–396 (2010)